МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп‘ютерної та програмної інженерії

Кафедра комп’ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____ А.С. Савченко
«___» _____ 20___р

ДИПЛОМНИЙ ПРОЕКТ

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ «БАКАЛАВР»

Тема: «Система обміну повідомленнями з обробкою в реальному часі»

Виконавець: студент УС-411 Пухальський Богдан Мирославович
(студент, група, прізвище, ім'я, по батькові)

Керівник: к. т. н., професор Віноградов Микола Анатолійович
(науковій ступень, вчене звання, прізвище, ім'я, по батькові)

Нормоконтролер: ст. викл. Шевченко О.П.
(П.І.Б.) (підпис)

КИЇВ 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп‘ютерної та програмної інженерії

Кафедра комп’ютерних інформаційних технологій

Освітній ступінь: Бакалавр

Галузь знань, спеціальність, спеціалізація: 12 “Інформаційні технології”, 122 “Комп’ютерні науки”, “Інформаційні управляючі системи та технології”

ЗАТВЕРДЖУЮ
Завідувач кафедри

А.С. Савченко
“_____”_____ 2021 р.

ЗАВДАННЯ

на виконання дипломного проекту студента

Пухальський Богдан Мирославович

1.Тема проекту: «Система обміну повідомленнями з обробкою в реальному часі» затверджена наказом ректора № 636/ст. від 22.04.2021р.

2. Термін виконання роботи: з 19.04.2020 по 11.06.2021р.

3. Вихідні дані до роботи: розробка високонавантаженої системи обробки подій із використанням брокера повідомлень Apache Kafka.

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці): вступ, аналітичний огляд і постановка завдання, розгляд завдань системи реального часу, дослідження технологій та засобів, розробка програмного продукту обробки потокових даних, оцінка продуктивності платформи, висновки.

5. Перелік обов’язкового графічного матеріалу: загальний перелік існуючих систем та обробка потокових даних створеним програмним продуктом. Використання структури проблематики систем реального часу.

КАЛЕНДАРНИЙ ПЛАН

	Етапи виконання дипломної роботи	Термін виконання етапів	Примітка
1	Аналіз літератури та джерел за темою дипломного проекту.	19.04.2021р.– 20.04.2021р.	
2	Розробка та затвердження плану дипломного проекту.	21.04.2021р.	
3	Проведення консультації з науковим керівником щодо створення першого розділу.	22.04.2021р.	
4	Аналітичний огляд і постановка задачі.	23.04.2021р.– 28.04.2021р.	
5	Порівняльний аналіз альтернативних засобів обміну інформацією	29.04.2021р.– 01.05.2021р.	
6	Аналіз систем реального часу	02.05.2021р.– 06.05.2021р.	
7	Розробка високонавантаженої системи обробки подій із використанням брокера повідомлень Apache Kafka та її аналіз продуктивності.	07.05.2021р.– 03.06.2021р.	
8	Висновки та оформлення пояснювальної записки дипломного проекту.	04.06.2021р.– 07.06.2021р.	
9	Підписання необхідних документів у встановленому порядку.	08.06.2021р.– 09.06.2021р.	
10	Підготовка до захисту та попередній захист дипломного проекту на випусковій кафедрі дипломного проекту	10.06.2021р.– 11.06.2021р.	

Студент **(*Пухальський Б.М.*)**

Керівник дипломної роботи (***Віноградов М.А.)***

РЕФЕРАТ

Пояснювальна записка до дипломного проекту «Система обміну повідомленнями з обробкою в реальному часі» містить: 57 сторінок, 11 рисунків, 30 літературних джерел.

Об’єкт дослідження: розробка систем програмного реального часу

Предмет дослідження: брокер повідомлень із відкритим програмним кодом Apache Kafka.

Мета роботи: визначення доцільності використання брокеру повідомлень у програмних системах реального часу на прикладі розробленої платформи

Методи дослідження, технічні та програмні засоби: розробка програмного забезпечення, випробування системи на навантаження, порівняльний аналіз, обробка літературних джерел.

Отримані результати та їх новизна: створено систему замовлення вантажних авіаперевезень із мікросервісною архітектурою, комунікація між якими відбувається централізовано через Apache Kafka. Проведено випробування на навантаження та отримано графіки зміни критеріїв продуктивності з часом. Платформу можна використовувати як сторонню систему для взаємодії з користувачами, аналітики та управління даними.

БРОКЕР ПОВІДОМЛЕНЬ, СИСТЕМА РЕАЛЬНОГО ЧАСУ, МОДЕЛІ ДАНИХ, МІКРОСЕРВІСИ, APACHE KAFKA.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 6
ВСТУП 7
РОЗДІЛ 1. ОГЛЯД ТЕПЕРІШНЬОЇ СИТУАЦІЇ У ОБЛАСТІ ОБМІНУ ПОВІДОМЛЕННЯМИ 9
1.1. HTTP протокол 9
1.2. Використання спільної бази даних 11
1.2.1. Реляційні бази даних 12
1.2.2. Документно-орієнтовані бази даних 13
1.3. Використання спільного сховища 15
1.3.1. Об’єктне сховище 15
1.3.2. Блокове сховище 16
1.4. Системи обміну повідомленнями із використанням JMS API 17
1.5. Висновки до розділу 19
РОЗДІЛ 2. МЕТОДИ ОБРОБКИ ПОВІДОМЛЕНЬ В РЕАЛЬНОМУ ЧАСІ 20
2.1. Специфіка технічних систем реального часу 20
2.1.1. Системи програмної обробки в реальному часі 21
2.1.2. Системи апаратної обробки в реальному часі 23
2.2. Розробка систем реального часу із застосуванням Apache Kafka 24
2.2.1. Архітектура Apache Kafka 25
2.2.2. Переваги над аналогами 27
2.2.3. Недоліки та необхідності модифікації 29
2.3. Висновки до розділу 31
РОЗДІЛ 3. ПЛАТФОРМИ ІЗ ЗАСТОСУВАННЯМ KAFKA 32
3.1. Розробка платформи аналізу виконання оптимальних вантажних авіаперевезень 33
3.2. Випробування на навантаження розробленої платформи 47
3.3. Висновки до розділу 51
ВИСНОВКИ 52
СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ 55

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

HTTP — протокол передачі гіпертексту

AWS — веб сервіси надані компанією Amazon

JMS — API для передачі повідомлень у промислових масштабах

API — програмний інтерфейс застосунку

DDOS — розподілена відмова системи

AMQP — розширений протокол чергування повідомлень

СУБД — система управління базами даних

ВСТУП

Обмін повідомленнями — важливий аспект нашого життя. Ще з давніх-давен люди передавали одне одному інформацію різноманітними способами. Використовувались вербальні методи, письмові (лист, телеграма) та візуальні (карти, схеми, тощо). Сповіщення даних не втрачає свою актуальність і сьогодні. Ми часто дізнаємось про актуальну інформацію зі стрічки новин, служби миттєвих повідомлень та різного роду сповіщень.

У наш час, коли популяція зростає та людство розвивається семимильними кроками, важливою є саме швидкість обробки та доставки даних в кінцеву точку. Світ змінюється швидко і для того, щоб людині вижити в цьому світі, потрібно іти в ногу з часом, а саме: отримувати медичним працівникам актуальні дані про статистику захворюваності, стан пацієнтів та статус доставки вакцин. У вік інформаційних технологій швидкодія — це не лише конкурентоспроможність, але й необхідність.

Актуальність. Сучасні інформаційні системи надають можливість парнійної обробки даних — завантаження вхідних даних кортежами великих об’ємів, кількість, яких налічує мільярди. Зазвичай вони записуються у структурований файл типу *.csv, *.json, *yml та завантажуються на сервер для подальшої обробки, агрегації, аналізу та конверсії. При такому підході актуальною залишається проблема продуктивності даної системи. Досить часто вона вирішується індексуванням та розділенням таблиці бази даних для прискорення зчитування інформації, але при цьому сповільнюється її запис.

Крім того, це ще породжує проблему зберігання даних. Об’єм їх стрімко зростає, так як все більше і більше людей користуються інформаційними системами та все більше “інформаційних слідів” зчитується. Дані великих розмірів вимагають багато простору для зберігання. Зазвичай дане питання вирішується зжаттям даних, але швидкість їх зчитування суттєво погіршується так як інформація перед зчитуванням чи коректним відображенням повинна бути

розпакована. Метою даної роботи є вибір та впровадження система обміну повідомленнями.

Досягнення мети потребує наступних дій:

1. Дослідити предметну область.
2. Розглянути методи та засоби, що застосовуються при передачі повідомлень миттєво та без втрат між мікро сервісами.
3. Скласти порівняльну характеристику переваг та недоліків різноманітних брокерів повідомлень та альтернативних методів комунікації мікросервісів (протокол HTTP, спільна база даних, спільне сховище).
4. Впровадити обраний метод у інформаційну систему, яка аналізує та агрегує дані в реальному часі.

Об’єкт проектування – визначення геолокації, виконання аналізу виконання оптимальної доставки та конверсії за допомогою підключення системи обміну повідомленнями з використанням Java Messaging System Application Programming Interface(далі JMS API).

Предмет проектування — методи обробки даних в реальному часі та без втрат.

Галузь застосування. Досліджені та застосовані методи обміну інформацією можна використовувати при розробці системи для обробки та аналізу даних в реальному часі та їх передачі без затримок та втрати. Практична цінність використовуваних методів полягає у визначенні оптимальних методів обробки та передачі інформації для конкретної системи. Дослідження включають приклад моделі даних за якими інформація передається без затримок та втрат.

РОЗДІЛ 1
ОГЛЯД ТЕПЕРІШНЬОЇ СИТУАЦІЇ У ОБЛАСТІ ОБМІНУ ПОВІДОМЛЕННЯМИ

1.1. HTTP протокол

HTTP (Hyper Text Transfer Protocol (англ. Протокол передачі гіпертексту)) протокол для дистрибутивних систем, що застосовують гіпермедіа. Даний протокол є базовим для передачі даних у всесвітній мережі, де гіпертекстові документи включають гіперпосилання на інші ресурси, до яких користувач може легко доступитися, такими способами як, натисканням миші або дотиком до екрану у веб браузері. HTTP належить до протоколів прикладного рівня моделі OSI[1].

Кожен HTTP запит може містити URL, заголовки та тіло. URL - Uniform Resource Locator(англ. Уніфікований покажчик інформаційного ресурсу) Кожен URL містить доменне ім’я та парметри. Доменне ім’я являє собою набір фізичних машин, через які потрібно здійснити запит для доступу до бажаного ресурсу. Параметри — вхідні дані, що задаються при виклику заданої точки повернення. Дана конфігурація доступна тільки для зчитування, значення їх змінити не можна. Параметри надсилаються на сервер у вигляді пар ключ-значення, які містить URL[3].

Заголовок — додаткова інформація, що надсилається разом із запитом. Зазвичай заголовки використовують для передачі секретної інформації такої як токен авторизації JWT[4].

Тіло — додаткова інформація, що надсилається разом із запитом та може бути представлена у форматованому вигляді. Зазвичай у тіло записують деякі дані, які є структурованими по специфікації JSON, XML або YAML. Прикладами такої інформації може бути нова сутність, яку потрібно записати у базу, список об’єктів для обробки на сервері та інше. Після надсилання запиту приходить відповідь, структура якої містить тіло та заголовки.

HTTP запити можна здійснювати різноманітними методами. Всього їх 9: GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, PATCH [2]. Кожен з цих методів призначений для виконання лише однієї операції над ресурсом. Дані методи можуть володіти такими властивостями як ідемпотентність — повернення одного і того ж результату при кожному виклику на одних і тих же параметрах та безпечність — здатність не змінювати представлення об’єкта.

Метод GET слугує для запиту представлення ресурсу. Запити таким методом призначені лише для отримання інформації про певний ресурс. Запит методом GET містить лише URL та заголовки. Метод є ідемпотентним та безпечним.

Метод HEAD за своєю функціональністю подібний до GET при виключенні отриманої відповіді, яка приходить без тіла. Метод є ідемпонетним та безпечним.

POST метод використовується для відправлення сутності на заданий ресурс, часто спричиняє зміну стану сутності або побічні ефекти на стороні сервера. Навідміну від методів згаданих вище, метод POST містить тіло запиту, куди можна записати об’єкт для відправки у формі звичайного тексту, JSON, XML, YAML та інше. Даний метод є ні ідемпотентним, ні безпечним так як впливає на зміну ресурсу.

Метод PUT замінює всі поточні представлення цільового ресурсу із запитом корисного навантаження. Даний метод можна розглядати як POST який, не записує новий ресурс на бекенд, а змінює існуючий. Так як і POST, PUT не є безпечним, але натомість є ідемпотентним.

DELETE метод видаляє заданий ресурс. Структурою цей метод подібний до GET та POST, тобто не містить тіла запиту. Володіє властивостями ідемпотентності, так як результат видалення кожного разу сталий, але не є безпечним через те, що

представлення об’єкта змінюється затиранням.

CONNECT встановлює тунель на сервер, який є визначеним цільовим ресурсом. Структура даного методу не містить тіла запиту. Метод не володіє жодною з властивостей притаманною HTTP запитам.

OPTIONS використовується для того, щоб описати з’єднання із цільовим ресурсом. Запит методом OPTIONS не містить тіла. Із властивостей HTTP запитів притаманна як ідемпотентність, так і безпечність.

TRACE виконує тестування повернення повідомлення вздовж повного шляху до цільового ресурсу. Структурою та властивостями подібний до метода OPTIONS.

PATCH використовується для застосування часткового оновлення цільового ресурсу. Даний метод можна розглядати як частковий PUT. Структура запиту методом PATCH містить тіло. Даний метод не є ні ідемпотентим, ні безпечним.

Завичай протокол HTTP використовується для реалізації Representational State Transfer (далі REST API), який дозволяє нам виконувати операції створення, зчитування, оновлення, видалення та багато інших за допомогою HTTP запитів[5]. Використовуючи властивості HTTP запитів такі як ідемпотентність, безпечність та кешування ми можемо налаштувати програмний інтерфейс системи з гнучким керуванням. Ще однією перевагою HTTP є здатність сповіщати користувача про поточний стан системи за допомогою кодів відповіді[6].

Не варто забувати і про недоліки розглянутого протоколу. Набір даних, який передається по HTTP, неможливо передати в 2 чи більше потоків і це суттєво погіршує швидкість отримання відповіді. Другим недоліком є втрата даних. Досить часто, при передачі даних на сервер великих об’ємів можна отримати помилку типу 5xx. Частота надсилання запитів обмежується для уникнення DDOS атак, запобіганню перевантаження сервера та інших причин.

1.2. Використання спільної бази даних

Бази даних стали невід’ємною частиною нашого повсякденного життя. Тому обговорення цієї теми ми розпочнемо із розгляду бази даних як деякий набір

зв’язаних даних та системи управління базами даних або СУБД як програмне забезпечення, яке керує доступом до бази даних.

1.2.1. Реляційні бази даних

На сьогоднішній день реляційні СУБД стали домінуючим типом програмного забезпечення для обробки даних. Дане програмне забезпечення представляє собою друге покоління СУБД, що засновується на використанні реляційної моделі даних, запропонованої Е.Ф.Коддом в 1970 році [7]. В реляційній моделі всі дані логічно структуровані всередині відношень (таблиць). Кожне відношення має ім’я та складається із іменованих атрибутів (стовпців) даних. Кожен кортеж (рядок) даних містить по одному значенню кожного з атрибутів (див. рис. 1.1).

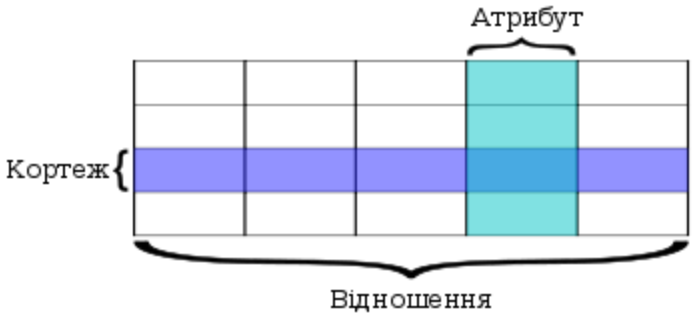

Рис. 1.1. Структура відношення у реляційних базах даних

Велика перевага реляційної моделі полягає саме в цій простоті логічної структури, яка запобігає дублюванню даних. Хоча, за цією простотою приховується сер’йозний теоретичний фундамент, якого не було у попередніх поколінь СУБД. Реляційна модель заснована на математичному понятті відношення, фізичним представленням якого є таблиця. Справа в тому, що Кодд будучи досвідченим математиком, широко використовував математичну термінологію, особливо із теорії множин і логіки предикатів.

Реляційні бази даних широко застосовуються у багатьох сферах для зберігання фінансових даних, ведення обліку інвентарю, зберігання даних про користувачів,

тощо. Хоча вони і є досить популярними, їм властиві деякі недоліки. Більшість серверних мов програмування таких як Java, C#, C++ використовують об'єктно-орієнтований підхід, в той час як дані в базі структуровані за логічною моделлю сутність-зв'язок. Вибірку отриману з бази завжди потрібно перетворювати в об'єкт для його подальшої обробки на сервері і це займає деякий час для реалізації подібного перетворення та роботи компонента програми. Хоча дана проблема вирішується інструментом перетворення відношення у об'єкт, але застосування такого автоматизованого підходу породжує і інші проблеми такі як N+1[8].

Досить жорстка структура реляційної моделі бази даних робить базу ізольованою при цьому інформація, яка у ній зберігається не може передаватися між системами з легкістю. У кожній системі реалізована унікальна бізнес логіка, яка спонукає до моделювання унікальної моделі даних, яка може бути застосована тільки у даній системі. Для передачі на іншу систему дані потрібно перетворити, що можливо не завжди. Саме тому у мікросервісних архітектурах, завичай, до однієї бази під'єднаний лише один сервіс.

Реляційні бази даних розроблені для того, щоб впорядковувати дані за певними характеристиками. Слідуючи такій специфікації, ми з легкістю можемо впорядкувати інформацію про сутності, такі як: користувачі, товари, підприємства, тощо, але такий підхід не є оптимальним для запису складних зображень та інших мультимедіа продуктів. Зазвичай потоковими даними, що обробляються в реальному часі є відео та аудіо потоки, що використовуються у системах для трансляції подій, тому використання спільної реляційної бази даних не є рішенням при передачі такої інформації.

1.2.2. Документно-орієнтовані бази даних

Документно-орієнтовані бази даних відносяться до так званого класу NoSQL, тобто, не вимагають мови запитів для роботи з нею так як дані у них не структуровані за реляційною моделлю[9]. Такого типу база даних являє собою набір документів. Кожному документу виставлено у відповідність ключ, за яким можна

його ідентифікувати та знайти. Кожен документ у базі даних розглядається як таблиця у реляційних базах даних, але ці документи ніяк між собою не зв’язані. Зазвичай дані структуруються у файлах формату *.json, *.xml чи *.yml. Найбільш популярними документно-орієнтованими базами даних є Elasticsearch, MongoDB та Amazon DynamoDB.

Дані у документно орієнтованих базах даних, як було згадано вище, зберігаються у файлах структурованого типу. Інформація записується у формі деревоподібної структури даних, що дозволяє реалізовувати пошук з логарифмічною алгоритмічною складністю за замовчуванням, на відміну від реляційних де така швидкодія можлива лише при виставлених індексах. До того ж, набір даних, на яких здійснюється пошук зберігається в одному і тому ж місці. Це вирішує проблему швидкодії при нормалізації даних.

У реляційних базах даних поле має існувати для кожної частини інформації та при кожному входженні. Якщо ж інформація недоступна, комірка лишається порожньою, але вона все ще має бути присутньою. Документно орієнтовані бази даних в цьому плані є більш гнучкими, структура окремого документу не повинна бути цілісною. Навіть неструктуровані дані великих об’ємів можуть бути збереженими у базі даних.

Структура даних документно орієнтованої моделі є досить гнучкою. До кожного ключа можна прив’язати файл унікального типу в контексті бази. Це суттєво спрощує подальше розширення структури даних та дозволяє зберігати інформацію у тому чи іншому вигляді відповідно до різноманітних потреб.

Одним із недоліків документно орієнтованої бази даних є неможливість зберігати дані які вимагають структуризації по складній моделі, такій як реляційна. При реалізації моделі сутність-зв’язок, дані, які описують сутність одного і того ж типу ми записуємо в одну таблицю, дані, які ми хочемо “приєднати” — в іншу. Звісно, у користувацькій системі можна реалізувати зв’язки між документами, але при цьому, система стає досить громіздкою та складною в реалізації та використанні. Саме тому, в документно орієнтованих базах даних зберігають лише інформацію, котра не вимагає складної структуризації, таку як події, що містять

лише ключ, мітку часу та коротке повідомлення.

1.3. Використання спільного сховища

1.3.1. Об’єктне сховище

Об’єктне сховище — це архітектура яка спроектована для обробки неструктурованих даних великих об’ємів. Це дані, які не підлягають організації в модель сутність-зв’язок, як у реляційних базах даних. На сьогоднішній день, більшість інформації, що поширюється в інтернеті є не структурованою, включаючи електронні листи, відео, фото, веб=сторінки, аудіофайли, дані з датчиків та інші типи даних як текстові, так і не текстові. Ці дані постійно надходять із соціальних мереж, пошукових рушіїв, мобільних гаджетів та пристроїв підключення до інтернету речей [10].

Маркетингові дослідження стверджують, що не структуровані дані з великою ймовірністю займуть частку 80% до 2025 року. Об’єкти є дискретними елементами даних, що зберігаються у структурно простому середовищі. Таке сховище не використовує папок, директорій та інших складних ієрархій, які є присутніми у файловій системі. Кожен об’єкт є простим, самодостатнім репозиторієм з даними, мета даними та унікальним ідентифікаційним номером.

Об’єкти в об’єктно-орієнтованій системі є доступними через API. Нативним API об’єктно-орієнтованого сховища є REST API, про який було згадано раніше. Цей інтерфейс запитує метадані об’єкта, для запису вихідного об’єкта через інтернет з будь-якої точки або будь-якого пристрою.

Однією із суттєвих переваг об’єктно-орієнтованого сховища є масштабованість, яка дозволяє без обмежень додавати пристрої та сервери, що дозволяє збільшувати об’єм сховища для зберігання даних все більших об’ємів та підтримувати вищу пропускну здатність при обробці даних.

Об’єктне сховище є простим у користуванні на відміну від файлових систем та реляційних баз даних. Спрощена ієрархія забезпечує вищу швидкодію при операціях

зчитування та запису.

Об’єктне сховище без проблем інтегрується з хмарною системою, яка надає багатокористувацьку систему для підприємств. Це дозволяє багатьом компаніям чи департаментам розділяти один репозиторій де кожен має доступ до виділеної партиції даних. Такий підхід оптимізує масштабування та витрати. Із вище перерахованого можна зробити висновок, що об’єктно орієнтоване сховище є оптимальним для використання моделей даних, які не вимагають перетворення для подальшої обробки в системі. На жаль, даний тип сховища не дозволяє розпаралелити передачу інформації для прискорення передачі, що є критичним для потокової роботи даних великих об’ємів.

1.3.2. Блокове сховище

Блокове сховище іноді розглядається як як сховище блокового рівня — це технологія, яка використовується для зберігання даних у вигляді файлів у мережі призначеній для сховища або у хмарно подібних середовищах для зберігання. Розробники використовують блокове сховище у тих випадках коли їм потрібна швидка, ефективна та надійна передача [11].

Блокове сховище розбиває дані на блоки та зберігає їх як окремі частини, призначивши кожній унікальний ідентифікаційний код. Мережа-сховище розміщує ці блоки даних у призначене їм місце. Це означає, що воно може зберігати ці блоки на будь-якій системі та кожен блок може бути сконфігурованим або розділеним для роботи з різними операційними системами.

Перевагою блокового сховища є те, що воно виокремлює дані від середовища користувача, завдяки цьому, дані можуть бути поширеними через різні середовища. Це створює безліч шляхів для доступу до даних та дозволяє користувачу швидко їх зчитувати, на відміну від об’єктного сховища, робота з яким можлива лише через API який не є розробленим для усіх мов програмування.

Блокове сховище є розділеним, що унеможливлює його оптимальне використання. Цю ситуацію, звісно можна виправити регулярною дефрагментацією,

але так як операція дефрагментації являє собою цикл зчитувань/записів — це прискорює знос фізичних накопичувачів, на яких інформація зберігається.

1.4. Системи обміну повідомленнями із використанням JMS API

Вводячи невеликий дискурс в історію, варто пригадати, що протягом тривалого часу і до сьогодні розробляються програмні продукти, які зберігають інформацію у базах даних. Зараз бази даних спонукають нас думати про світ у контексті речей, як користувачі, замовлення, літак, тощо. Кожна з подібних речей перебуває у деякому стані. Ми отримуємо цей стан та записуємо його до бази даних. Це працювало досить добре протягом десятиліть, але зараз стало прийнятим вважати, що в першу чергу варто думати за події, а потім за речі.

Перш за все події теж перебувають у деякому стані. Кожна подія містить опис того, що сталося, але основна ідея полягає в тому, що подія — це ідентифікація в часі, коли сталася зміна із сутністю. Зберігання подій в часі є досить громіздким. Для виконання цієї задачі ми використовуємо таку структуру як журнал — впорядкована послідовність подій. Як тільки з'являється подія, ми записуємо її в журнал включаючи стан, опис та час у який відбулась дана подія. Неважко припустити, що журнали є простими для розуміння а також зручними для проектування, реалізації та масштабування, що є протилежним для баз даних. Історично склалося, що операції масштабування є досить громіздкими для них.

Сервіс обміну повідомленнями — це система управління даними журналами. В контексті термінології таких систем перевалочний пункт для відправки журналів називається темою. Тема — це впорядкована колекція подій, які зберігаються протягом тривалого часу. Тут мається на увазі, що вони записуються на диск та копіюються, тобто, зберігаються на більш ніж одному диску та більш ніж одному сервері, на якому працює інфраструктура. Таким чином, унаслідок апаратного виведення з ладу дані не втрачаються, що суттєво скорочує втрати при передачі даних. Теми можуть зберігати дані як і протягом коротких періодів (декілька годин, днів) так і протягом тривалих (декілька чи навіть сотню років). Також дане сховище

може бути як малих об’ємів, так і великих.

Такі характеристики є сприятливими для розробки різноманітних систем в контексті використання ресурсів та архітектури. Кожна подія записана у тему являє собою будь яку дію, що відбувається у реальному часі, таку як: записати користувача, користувач обновив адресу доставки, посилки завантажують у літак або показники термометра змінились. В даній ситуації, кожна із цих речей може бути подією, що зберігається у темі і сервіс обміну повідомленнями залучає вас думати про події в першу чергу та речі — потім.

Заглядаючи у минуле, коли бази даних були найпоширенішим рішенням для зберігання, було досить популярним будувати єдину велику програму, яка використовує велику єдину базу даних (моноліт) і це вважалося звичним по ряду причин, але це доходило до того моменту, коли внесення змін ставало дедалі важчим та , власне, їх розробка. Такі системи виростали до величезних масштабів та кожному розробнику ставало все важче утримувати всі деталі в голові. Кожен інженер, який коли-небудь працював над розробкою подібних систем, доведе, що це — правда. В наш час досить популярною є розробка архітектури, що складається із низки малих сервісів (мікросервісів), кожен із них достатньо малий для сприйняття, незалежної нумерації версій, зміни, індивідуального покращення і ці сервіси можуть комунікувати між собою за допомогою тем. Таким чином кожен із сервісів може зчитувати повідомлення із теми, виконувати обчислення над ними та відправити результат до іншої теми. Тепер результат тривалий та записується для постійного зберігання іншими сервісами та концернами в системі для подальшого виконання.

Недоліком тем сервісів обміну повідомленнями є складність налаштування та розгортання. Дана система є досить складною для розуміння, адже кожна тема поділена на розділи. Розділи дозволяють поділити дані та відправити їх у декілька потоків для прискореного надсилання чи одночасного сповіщення багатьох систем, які слухають цю тему. При даному підході варто оперувати складними поняттями багатопотокової обробки даних та запобігати поширеним проблемам як “перегони потоків” та “мертвий блок”[12]. Також потрібно налаштовувати політику повторного надсилання даних при не успішній обробці слухачем, політику

безвідмовної роботи системи при аварійній зупинці одного із слухачів та багато іншого.

1.5. Висновки до розділу

Проведена порівняльна характеристика методів обміну інформацією між системами, а саме протокол HTTP, використання спільної бази даних, використання спільного сховища та система обміну повідомленнями на базі JMS API. До уваги брались такі критерії як швидкодія, зручність та можливість використання, передача інформації без втрат та можвивість структурування даних різноманітними способами.

В даній роботі досліджується можливість передачі інформації моментально та без втрат, а також, її подальша обробка у режимі реального часу. Найменш придатним є спосіб використання спільної реляційної бази даних, так як у ній можна зберігати лише структуровані дані, які не підлягають простому масштабуванню та швидкому зчитуванню. Найбільш оптимальним вирішенням нашої проблеми є використання системи обміну повідомленнями на базі JMS API. Вона дозволяє передавати миттєві повідомлення без втрат, але можлива лише для передачі подій та є складною у налаштуванні.

РОЗДІЛ 2
МЕТОДИ ОБРОБКИ ПОВІДОМЛЕНЬ В РЕАЛЬНОМУ ЧАСІ

2.1. Специфіка технічних систем реального часу

Система реального часу — це система у якій відповідь має бути гарантована у визначений час або система яка виконує задачу у заданий термін [13]. До прикладу, система управління польотом, відображення метрик та інше.

Системи реального часу поділяють на 2 типи: системи програмної обробки в реальному часі та системи апаратної обробки в реальному часі.

Система програмної обробки в реальному часі - це система до вимог якої не входить виконання задач у заданий термін при виникненні певних умов та з дотриманням високої ймовірності. Не виконання задачі у таких системах у заданий термін не несе в собі катастрофічних наслідків. Практична цінність результатів отриманих у програмній системі обробки в реальному часі поступово спадає зі зростанням запізнення. Приклад: прогноз погоди.

Система апаратної обробки в реальному часі — тип системи реального часу, яка завжди виконує задачі у заданий термін. Виконання задач із запізненням у таких системах може призвести до трагічних наслідків. Практична цінність результатів отриманих системою апаратної обробки в реальному часі стрімко спадає і втрачає актуальність коли запізнення зростає.

Запізнення — це характеристика системи, яка визначає наскільки пізно система видає результати виконання задачі відповідно до заданих термінів. Приклад: апарат штучної вентиляції легень із автоматичним управлінням.

2.1.1. Системи програмної обробки в реальному часі

З появою інтернету та стохастичних мережевих систем, їх складність суттєво зросла та призвела до нетривіальних проблем у дослідженні та синтезу продуктивності системи. Протягом останніх років було здійснено дослідження питань високонавантажених мережевих систем, таких як відсутність вимірювання, зникнення сигналу, механізми комунікації що вступають в дію при виникненні події та довільно запущені кібератаки [14]. Дані дослідження подаються наступним чином.

Досить відомим є той факт, що традиційні схеми управління покладаються на припущення, що вимірювальні сигнали передаються ідеально. Таке припущення є досить консервативним, у багатьох інженерних практиках при представленні не надійних з’єднань комунікації. У практичній інженерії мережевих систем, визначені сигнали завжди є предметом ймовірної втрати даних(зникнення даних або втрата пакетів) із низки причин, таких як втрата сигналу, похибка у проектуванні, похибка датчика, перевантаження мережі та інших.

Існує два основних способи дослідження розподіленої системи на предмет збоїв — логування та трасування (стандарт open tracing)[15].

Логування — це додавання повідомлень у вихідному коді про ту чи іншу дію. До повідомлення автоматично прив’язується мітка часу коли ця подія відбулась та унікальний номер. Всі логи записуються у документно орієнтовану базу даних, наприклад ElasticSearch, фільтруються та агрегуються визначеними правилами у системі типу Logstash та відображаються у текстовому вигляді або у вигляді графіка у таких візуалізаторах як Kibana, Grafana та інших.

Трасування — це фіксування міток часу початку та закінчення атомарної дії з метою визначення шляху проходження запиту по веб вузлам. Кожен елемент траси

містить мітку часу до та після та метадані, які включають у себе назву сервісу, назву операції, тощо. При аналізі всі мітки збираються та відсортовуються для побудови діаграми. Найпопулярнішими інструментами трасування є Amazon X-Ray та ELK APM.

У мережевих системах використовуються контролери ініційовані подією та контролери ініційовані за розкладом. З технічної точки зору контролери ініційовані подією є більш цікавими для дослідження, особливо у розподілених системах реального часу та зменшення затримок при віддалених викликах. З іншого боку для продовження часу роботи системи та уникнення аварійної зупинки після перевантаження, слід зменшити частоту надсилання запитів на систему. Таким чином запити ініційовані подією є більш сприятливими для роботи розподілених систем.

Принцип дії систем, що запускають на вимогу полягає у використанні вже працюючих фізичних серверів. Збудований артефакт (архів бінарного коду із залежностями) прив’язується до системи запуску на вимогу, при надходженні запиту на систему, виділяються обчислювальні потужності на запущеному сервері, які, власне, і виконують запит. Недоліками таких систем є холодний старт (час на запуск системи перед обробкою запиту) та обмеження ресурсів таких як час виконання запиту на вузлі та оперативна пам’ять. Перевагою — економія ресурсів, так як вони використовуються лише за потреби. Прикладом такої системи є AWS Lambda.

Специфікою тестування таких систем є потреба в симуляції діяльності користувачів. Запуск контроллера ініційованого подією не можливо передбачити при тестуванні вживу, саме тому роботу системи потрібно ініціювати у потрібний для тестування час. Це зазвичай виконується інструментами для випробування навантаженням таких як Gatling чи Neoload.

Добре відомо, що датчики, контролери, та контрольовані заводи є часто приєднаними до мереж для систем мережевого управління. У такій ситуації, дані, що передаються без інформаційного захисту можуть з легкістю бути перехопленими зловмисником. Кібератака може бути розглянутою як метод, процес чи захід

зловмисної спроби зменшити надійність мережі або перехопити управління заводом. Відповідно до типу їх здійснення, вони поділяються на DOS (denial of service) атаки, атаки відтворення та атаки обману.

Варто відмітити точку зору захисту для заводів. Атаки позиціонують довільну поведінку так як успіх атак сильно залежить на можливості визначення захисного програмного забезпечення, протоколів комунікації та умов у мережі (тобто завантаженість мережі та швидкість передачі даних у мережі), які виникають у не визначений момент. До прикладу, хибні дані, надіслані ініціатором атаки обману можуть бути визначені при використанні деякого апаратного забезпечення, інструментів програмного забезпечення чи алгоритмів (таких як детекторів алгоритмічної складності $O(n^2)$) які призводять до не вдалих атак. До того ж, у протоколі багатопроменевої маршрутизації із схемою передачі секретної інформації (T, N), зашифроване повідомлення розділяється на N передач і таким чином воно може бути з легкістю розшифорованим для будь-якого зашифрованого повідомлення T та при будь якій кількості передач [16].

Спираючись на цей факт, процес Бернуллі чи процес Маркова із відомою статистичною інформацією був застосований для управління DOS атаками [17]. Для запобігання таким атакам використовують мережі розподілу даних [18], але вони не враховують транзитивну динаміку замкнутої системи через стохастичну особливість та сигнали інтерференції надіслані зловмисниками. Інший спосіб запобігання таким атакам це акумулювання змінної-лічильника для кожного авторизованого користувача, значення якому зберігається у спільному кеші, але використання таких ресурсів досить вартісне.

2.1.2. Системи апаратної обробки в реальному часі

Комп’ютерні системи реального часу вимагають подвійного тлумачення коректності: значення, яке потрібно визначити повинно бути не лише коректним, але і повинно бути визначеним у заданий час. У апратних системах реального часу, певні частини обчислень мають терміни асоційовані з ними і це є досить

імперативним для корекції такої системи, де всі ці частини обчислень завершують свою роботу у заданий час. У порівнянні програмні системи реального часу, відповідно до не жорстких вимог, в деяких випадках можуть іноді пропустити задані терміни виконання обчислень або не виконати обчислення у заданий час, але не більш ніж при заданому об’єму даних [19].

Однією із основних особливостей між теорією визначення задач реального часу та “традиційною” теорією визначення задач є те, що навантаження зазвичай характеризується як дещо згенероване скінченною колекцією рекурентних задач або процесів [20]. Кожна така рекурентна задача може моделювати частину програмного коду, який впроваджується у нескінченний цикл або є запущеним при виникненні деякої зовнішньої події, як було згадано вище. Кожне виконання програмного коду розглядається як робота, задача, в свою чергу, генерує нескінченну послідовність робіт.

Рекурентна задача вважається періодичною коли послідовні роботи задачі повинні бути згенеровані протягом певної тривалості та спорадичною коли задана лише нижня межа для генерації послідовних робіт. Саме такі задачі застосовуються у багатопроцесорних системах реального часу.

Різноманітні моделі запропоновані для представлення спорадичних задач, деякі із них широко поширені, включаючи модель Лю та Лайленда [21] та модель трьох параметрів [22].

Нещодавно, спільнота інженерів систем реального часу вирішила переосмислити, що типи навантаження визначені у типових системах реального часу характеризуються не за фактом типової генерації рекурентними задачами, але і позиціонуються внутрішньо задачним паралелізмом. Отже, були запропоновані моделі, що базуються на напрямленому, ациклічному графі для представлення рекурентних процесів, що входять до складу систем реального часу.

2.2. Розробка систем реального часу із застосуванням Apache Kafka

Перед інженерами часто стоїть задача побудувати систему для обробки подій

в реальному часі. Однією із проблем реалізації таких систем є необхідність обробки запитів по мірі надходження. Досить великих зусиль вимагає розробка програмного забезпечення із використанням сокетів, що відтворюються на стороні сервера із врахуванням усіх деталь та різноманітних випадків поведінки. Крім того, дана реалізація є досить типовою та використовується на регулярній основі у різноманітних системах. Цілком обґрунтованим, є використання готового рішення для отримання, розподілення, обробки, та відправлення запитів у системах із високим трафіком. Це дозволяє нам не витрачати часу на реалізацію низькорівневих, типових операцій та зосередитись на розробці самої бізнес логіки.

Для вирішення таких проблем Apache Kafka надає потоки зчитування і запису та Java API для їх програмного управління. За допомогою таких засобів ми маємо можливість надсилати оброблені дані великих об’ємів та отримувати їх у нашому сервісі без втрат.

2.2.1. Архітектура Apache Kafka

Apache Kafka — це розподілена система, що складається із серверів та клієнтів, які з’єднуються по високошвидкісному TCP протоколу [23]. Дану інфраструктуру можна розгорнути як вручну, на фізичних серверах, так і на віртуальних машинах, контейнерах, системах запуску на вимогу та хмарах.

Kafka працює на кластері із одного або багатьох серверів, які можуть бути поширюватись на декілька дата центрів чи зон доступності. Ці сервери формують рівень зберігання, який називається брокером. Інші сервери запускають Kafka Connect для постійного імпортування та експортування потокових даних для інтеграції Kafka із користувацькою системою такою як реляційна база даних чи інші кластери. Для реалізації критично важливого функціоналу, Kafka кластер широко масштабується та є стійким до відмов: якщо будь який із даних серверів аварійно завершує роботу, трафік перенаправляється на інші працюючі сервери для забезпечення постійної обробки без втрати даних.

Клієнти дозволяють розробнику імплементувати розподілене прикладне

програмне забезпечення та мікросервіси, що зчитують, записують та обробляють потокові дані паралельно, масштабовано та у відмовостійкій манері, навіть у випадках проблем у мережі чи в апаратному забезпеченні. Існує безліч клієнтів створених спільнотою. Вони є доступними для Java та Scala, включаючи високорівневі бібліотеки потоків Kafka для Go, Python, C/C++, а також REST APIs.

Найбільш вживаними APIs для потоків зчитування та запису Apache Kafka є продюсери та споживачі. Продюсер — це програмне забезпечення користувача, яке публікує (записує) події до Kafka, а споживачі є підписниками (зчитувачами) цих подій. У Kafka продюсери та споживачі є виокремленими одне від одного, що є ключовим елементом дизайну для забезпечення широкої масштабованості. До прикладу, продюсери не повинні чекати на споживачів.

Як було згадано у попередніх розділах, усі події записуються у теми. Тема є розподіленою, це означає що тема поділяється на деяке число розділів, що знаходяться на різних Kafka брокерах. Така розподілена модель даних є важливою для масштабованості, тому що це дозволяє користувацьким застосункам зчитувати та записувати дані із багатьох брокерів одночасно. Нова подія, насправді, додається тільки до одного розділу теми. Події з тим же ключем записуються у той же розділ і Kafka дає гарантії, що будь який споживач зчитає події у тому ж порядку, у якому вони були записані (рис. 2.1).

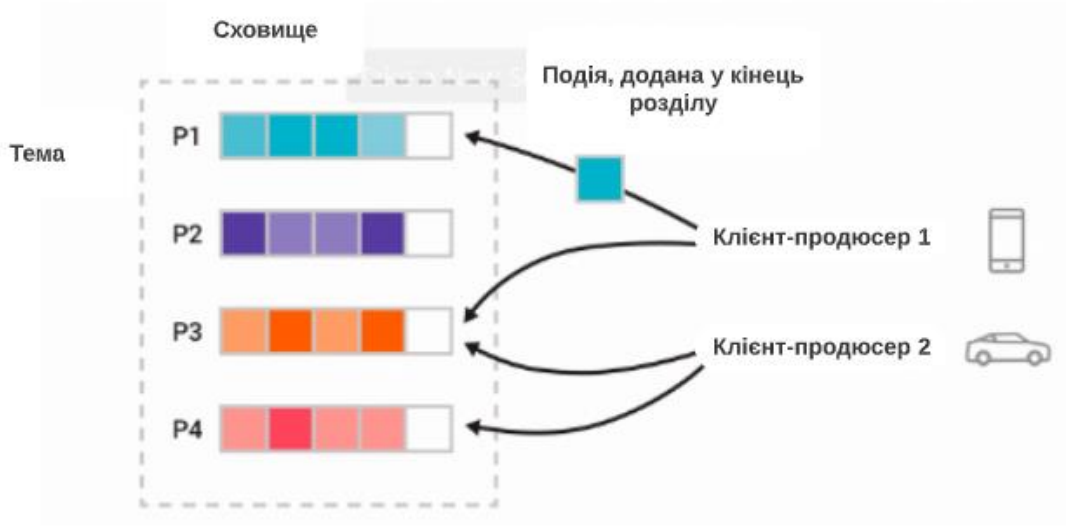

Рис. 2.1. Модель публікації подій та їх розповсюдження по розділам.

Для забезпечення відмовостійкості та легкої доступності даних, кожна тема може бути зреплікована, навіть через географічні регіони чи дата центри. Таким чином завжди існує деяке число брокерів із копіями даних для випадку неочікуваного виходу системи з ладу чи проведення планового випробування системи.

Apache Kafka надає 5 ключових APIs для Java та Scala [24]:

- Адміністративний — призначений для перевірки тем, брокерів, та інших об’єктів Kafka.
- Продюсер — призначений для публікації (запису) потокових даних у одну чи більше тем.
- Споживач — призначений для підписки (зчитування) потокових даних у одну чи більше тем.
- Потоковий — дозволяє реалізовувати застосунки для обробки потокових даних та мікросервіси . Він надає високорівневі функції для обробки потокових даних, включаючи трансформацію, агрегування, злиття та більше.
- Інтерфейс з’єднання — розроблений для побудови та запуску конекторів, які зчитують чи записують потокові дані у сторонні системи для інтеграції із Apache Kafka.

2.2.2. Переваги над аналогами

Зростаюча кількість обладнання, що під’єднується до мережі Інтернет призвела до нового поняття — інтернету речей. Дана система являє собою набір пристроїв, які здатні взаємодіяти між собою без втручання людини. Підтримка таких систем вимагає покращеної інтеграції і саме потреба у вдосконаленні автоматизованої комунікації між машинами призвела до створення брокерів повідомлень, таких як Apache Kafka. У даному розділі будуть розглянуті такі аналоги як RabbitMQ та Redis.

RabbitMQ є одним із перших брокерів повідомлень із відкритим програмним кодом та може бути використаним із багатьма мережевими протоколами. Від самого

початку дана система реалізовує протокол AMQP [25]. Цей протокол є призначеним для обміну повідомленнями із комплексним функціоналом маршрутизації. AMQP забезпечує гнучкість у використанні технологій брокерів повідомлень за межами екосистеми Java. Фактично RabbitMQ працює із Java, Spring, .NET, PHP, Python, Ruby та іншими мовами без проблем. Численні плагіни та бібліотеки також присутні.

Серед переваг даного брокера можна зазначити простоту використання (наданий як командний так і графічний інтерфейс), конфігурації, що може бути задана у коді та розгортання (присутні готові рішення такі як Spring Cloud). Основною перевагою Apache Kafka над RabbitMQ є можливість зберігати повідомлення протягом тривалого періоду. Іншою відмінною рисою є налаштування брокера, яке хоч і складніше та імперативніше, але більш гнучко дозволяє конфігурувати відмовостійкість та задавати правила для зберігання повідомлень при виникненні унікальних форс-мажорних ситуацій.

Redis (Remote Dictionary Server, англ. Віддалений сервер словника) — це не реляційна база даних із відкритим програмним кодом, яка, здебільшого, використовується як кеш для застосунку. Оскільки, у даній системі дані зберігаються у пам’яті із довільним доступом, це забезпечує вищу продуктивність, а саме коротший час отримання відповіді.

Перевагами Redis є висока швидкодія, що забезпечує надсилання будь яких даних, у тому числі потокових, у заданий час. Також, варто відмітити і просту структуру не реляційної бази даних, яка являє собою набір ключ-значення і дозволяє зберігати в собі та оперувати неструктурованими даними. Серед недоліків слід віднести обмеження на об’єм даних, котрі зберігаються. Якщо потрібно реалізувати систему, у котрій відбувається передача, аналіз та обробка даних великого об’єму у реальному часі із великим числом споживачів, варто використати брокер повідомлень Apache Kafka, так як він спроектований для зберігання великого об’єму даний та його кластер легко масштабується на декілька серверів чи навіть дата центрів.

Найдоцільнішим використанням Redis є кешування відповідей на запити.

Завдяки цьому, можна зменшити кількість віддалений викликів, в тому числі і на базу даних, та цим самим прискорити отримання відповіді на запит завдяки близькому фізичному розташуванню до кінцевого користувача.

2.2.3. Недоліки та необхідності модифікації

Apache Kafka включає в себе функції обміну повідомленнями, зберігання, та обробки. Це потужна платформа для обміну потоковими даними. Вона здатна зберігати історичні дані та працювати в режимі реального часу. Незважаючи на всі ці можливості, варто врахувати і деякі нюанси. Серед них: відсутність окремого ключа для кожного повідомлення, відсутність моніторингу підписників для кожної теми чи повідомлення, відсутність індексів і довільного доступу та очистка від не актуальних даних [26].

Ключ окремого повідомлення, яке записується до заданої теми визначає у який розділ необхідно розмістити дане повідомлення та, відповідно, який підписник його зчитає. Такий функціонал є необхідним і достатнім для передачі даних без втрат, але суттєво ускладнює моніторинг та налагодження високонавантаженої системи. Навіть у випадках, коли повідомлення було отримано певним сервісом-підписником, не завжди можливо обробити його відповідно до бізнес логіки, що призводить до утворення помилок та похибок у результатах обчислень. Для уникнення неочікуваної поведінки системи, варто проводити її виправлення, але перед цим необхідно провести аналіз шляхом перегляду журналу подій. Кожна подія в журналі окрім мітки часу повинна містити унікальний номер для її ідентифікації. Саме так можна визначити вхідні дані, які призвели до небажаної поведінки та обрати стратегію для її виправлення.

Незважаючи на постійну технічну підтримку веб вузлів, рано чи пізно будь який сервіс-підписник може відмовити. Причина такої відмови може бути різною: втрата з’єднання, закриття сокету, перевантаження сервісу та інше. Хоча, повідомлення з теми при цьому не видаляється, не всі споживачі зможуть отримати дані вчасно. Для покращення моніторингу споживачів варто запровадити набір

правил для сповіщення теми споживачем про стан повідомлення, такі як у RabbitMQ.

Тему можна розглянути як набір черг (розділів), що працюють по принципу “перший зайшов — перший вийшов”. Це дозволяє реалізовувати нам транзакції, де порядок виконання атомарних операцій є важливим. Іноді нам потрібно пропустити повідомлення, які не є актуальними, не відповідають призначенню підписника та з ряду інших причин, шляхом довільного доступу. Дану проблему можна вирішити на стороні продюсера. Для цього дані перед відправкою потрібно фільтрувати по критерію визначеним для певного підписника, але дане рішення є досить імперативним, вимагає великих зусиль та збільшує час на розробку.

Чудовою практикою є зберігання історичних даних, що не використовуються на даний момент. Вони дозволяють нам провести аналіз в майбутньому та на основі результатів прийняти рішення для вирішення тої чи іншої проблеми. Для більших даних потрібно більше простору для їх зберігання. Виділення додаткового простору є ресурсозатратним в плані фізичних накопичувачів та збільшує час зчитування та пошуку потрібних даних. Подібні проблеми вирішуються стисненням, яке хоч і економить простір, але час зчитування суттєво зростає, так як на розпаковку потрібен додатковий час. Іншим рішенням є заплановані фонові процеси, які видаляють дані, що були створені раніше заданого часу, але даний підхід є суттєвим лише для інформації, яка не підходить для аналізу.

Усі вищезгадані проблеми присутні у системі обміну повідомленнями Apache Kafka та призводять до зростаючого технічного боргу. На щастя, це — система із відкритим програмним кодом, що дозволяє вносити покращення не лише розробникам даного проекту. Іншою особливістю Kafka є присутність API для розробки плагінів, що розширюють основний функціонал системи та виправляють недоліки. Спільнота інженерів протягом останніх дев’яти років розробила безліч плагінів, які є придатними навіть для використання у промислових системах.

2.3. Висновки до розділу

В даному розділі надано опис систем реального часу. До опису входять визначення, що таке системи реального часу, та два основних типи, на які вони поділяються. Власне, двом типам систем реального часу присвячені підпункти першого підрозділу, у яких описуються особливості того чи іншого типу, принцип дії та проблеми, що досить часто виникають при розробці та експлуатації даних систем. Особлива увага приділялась найбільш критичним проблемам систем реального часу, які не можна упускати при розробці подібної системи, а особливо, при її першому розгортанні. Найбільш критичні проблеми можна умовно поділити на проблеми інформаційної безпеки, передача даних у заданий термін, що вирізняє систему реального часу з-поміж інших, та втрата даних. У даному дипломному дослідженні розглядаються дві останні із вищезгаданих проблем. За допомогою системи обміну повідомленнями на базі JMS API можна здійснити спробу вирішення проблем вчасної передачі даних та без втрат.

Ще одним призначенням даного розділу було дослідити переваги та недоліки Apache Kafka. На базі такого дослідження вирішено застосовувати саме таке рішення для передачі даних великих об’ємів у заданий час (до ста тисяч повідомлень за секунду). Звісно, як і у будь-якої системи, у Kafka присутній ряд недоліків, але їх легко можна виправити підключивши потрібний плагін

РОЗДІЛ 3
ПЛАТФОРМИ ІЗ ЗАСТОСУВАННЯМ KAFKA

Для визначення доцільності використання системи обміну повідомленнями Apache Kafka, було розроблено платформу із мікросервісною архітектурою. Метою застосунку є отримання запиту про замовлення авіаційних вантажних перевезень між вказаними пунктами та у вказану дату, а також, сповіщення користувача про можливість здійснення даної операції.

Дана система складається із чотирьох сервісів: сервіс замовлень, сервіс знаходження вантажу на складі, сервіс повідомлення погодних умов та сервіс оплати. Запит здійснюється від користувача до сервісу замовлень. Наступний сервіс, в свою чергу, отримує дані від інших трьох, агрегує та передає відповідь користувачу, у якій вказано наявність того чи іншого вантажу, можливість перевезення відповідно до погодних умов та терміни виконання замовлення.

Для зменшення кількості віддалених викликів, та цим самим, скорочення часу отримання відповіді користувачем, для з'єднання мікросервісів між собою застосовується брокер повідомлень Apache Kafka. Такий сервіс дозволяє отримати нам актуальні дані із усіх сервісів нашої системи і цим самим вирішити проблему не достовірних даних, які могли оновитися протягом синхронного виклику на один із сервісів.

3.1. Розробка платформи аналізу виконання оптимальних вантажних авіаперевезень

Розробка розглянутої платформи проводилась за допомогою мови програмування Java з бібліотеками JakartaEE та Apache Kafka Clients. Для збірки програмного коду у артефакт, було обрано інструмент Apache Maven. Даний інструмент збірки є декларативним та займає мінімум часу на налаштування. Хоча, на відміну від імперативних інструментів, його функціонал не є гнучким [27], але наші потреби він повністю задовольняє. В якості сервера-застосунку було обрано Apache Tomcat. Розглянутий проект містить дві версії: перша – із застосуванням Apache Kafka, друга – без. Для можливості підтримки двох версій одночасно та оперативного перемикання між ними було використано розподілену систему контролю версій Git.

Структура Git репозиторію являє собою набір гілок, що розгалужуються одна від одної[28]. Кожна гілка містить набір комітів, які є атомарними змінами коду. Для кожної версії продукту була виділена окрема гілка: kafka – для версії із застосуванням брокеру повідомлень та master (за замовчуванням) для версії із застосуванням HTTP. Функціонал розробленої системи представлено на UML діаграмі Use Case (рис. 3.1).

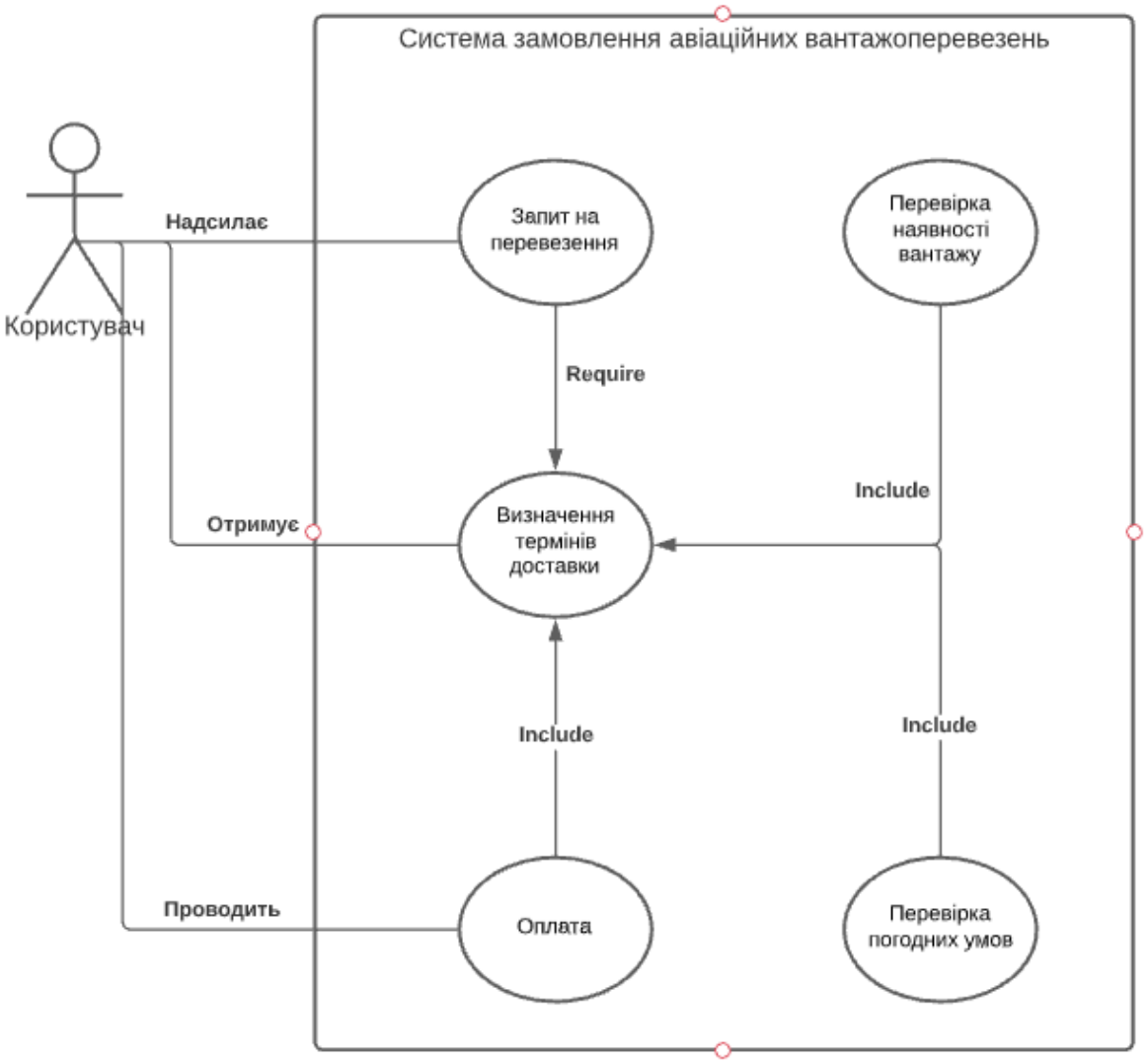

Рис. 3.1. Use Case діаграма системи вантажних авіаперевезень

При об’єктно-орієнтованому проектуванню системи було надано перевагу композиції над наслідуванням. Об’єкт, набуваючи властивостей батьківського об’єкта суттєво збільшується у розмірі. Це особливо помітно при реалізації громіздких систем. При композиції об’єкти розташовуються у пам’яті незалежно одне від одного та можуть використовуватись повторно різними елементами програми як залежності (рис. 3.2).

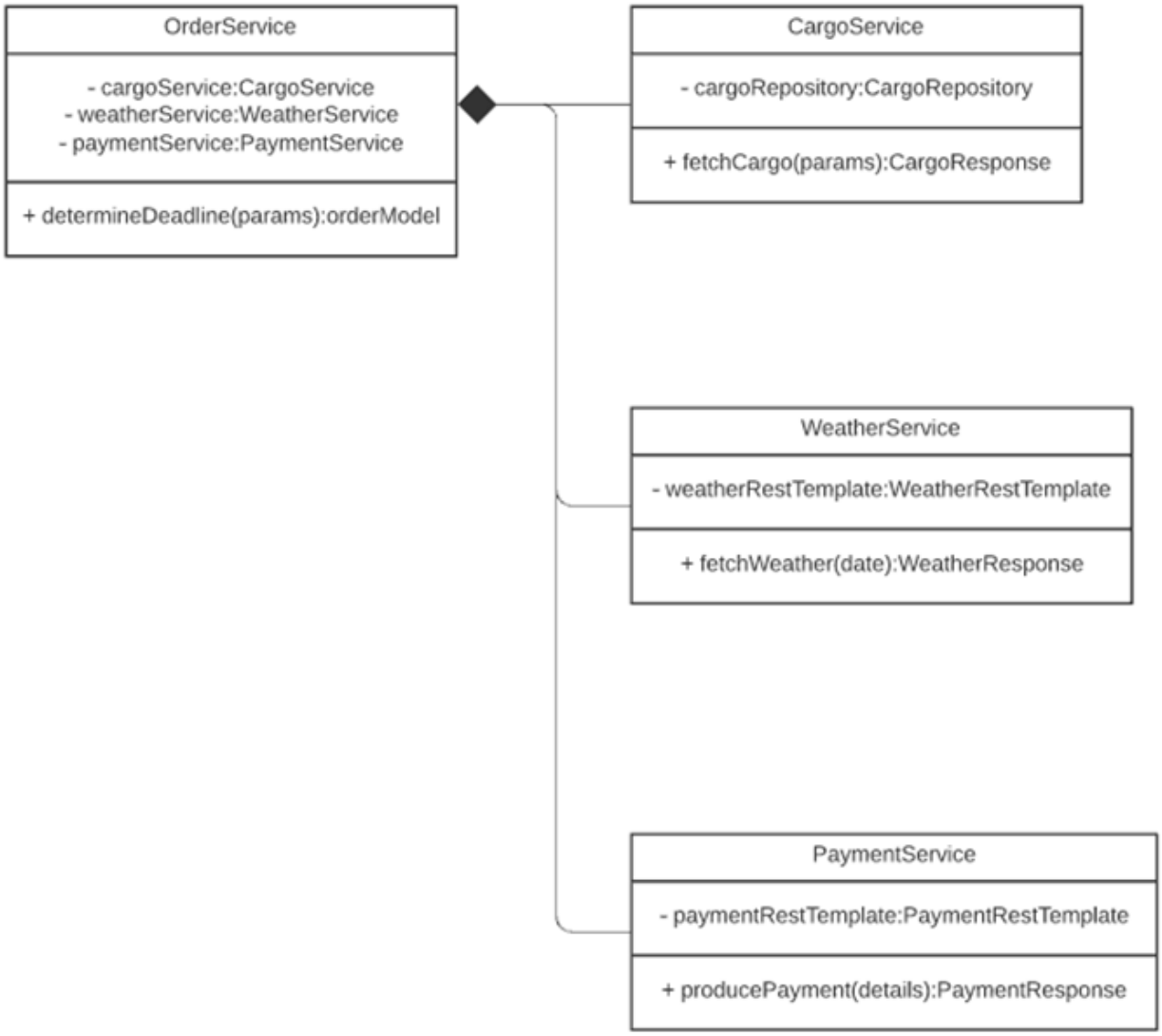

Рис. 3.2. Діаграма класів розглянутої системи

Як було згадано вище, система обміну повідомленнями Apache Kafka дозволяє нам здійснювати асинхронні виклики на різноманітні сервіси з метою збору даних. На UML діаграмі послідовностей представлено три виклики на CargoService, WeatherService та PaymentService, які можуть виконуватись одночасно протягом одного і того ж проміжку часу (рис.3.3).

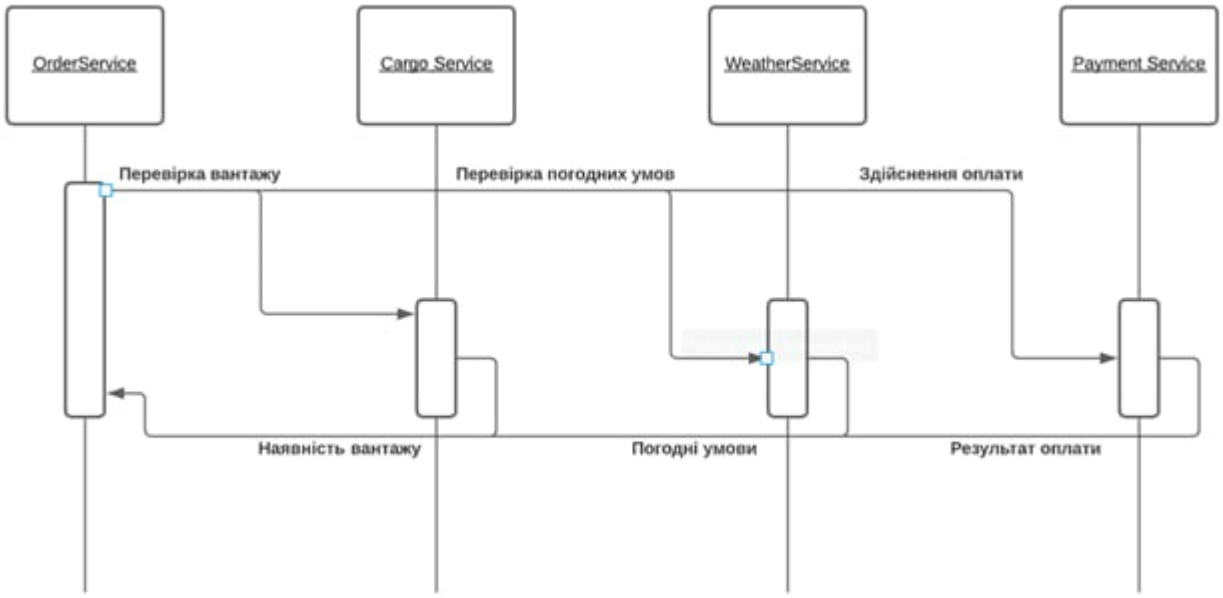

Рис. 3.3. Діаграма послідовностей розробленої системи

У першій версії розробленої системи виконуються синхронні виклики на допоміжні сервіси за допомогою протоколу HTTP (рис. 3.4).

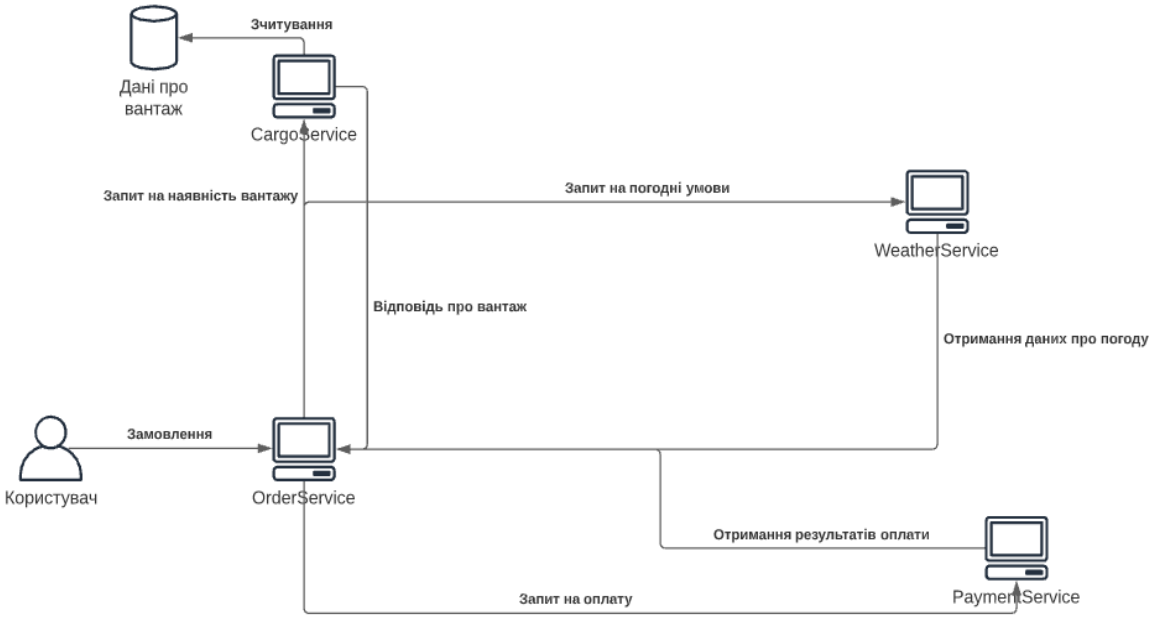

Рис. 3.4. Архітектура системи першої версії

У другій версії системи обмін даними між мікро сервісами відбувається централізовано через систему Apache Kafka. Як тільки дані по певному запиту

оновлюються на будь-якому із представлених сервісів, вони одразу ж записуються у тему брокера повідомлень. При зчитуванні запитаних даних споживач завжди отримує актуальну інформацію. Під час агрегації даних на сервісі замовлень інформація береться із повідомлень, відфільтрованих за номером замовлення (рис 3.5).

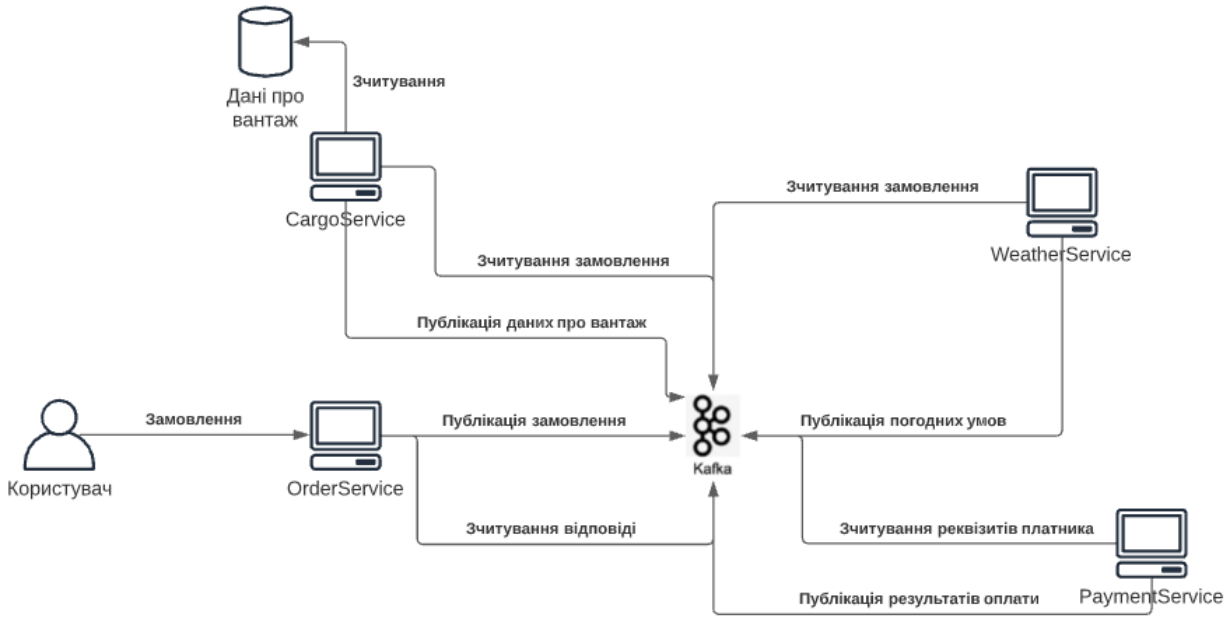

Рис. 3.5. Архітектура розробленої системи другої версії

Код програми:

```
package pukhalskyi.cargoservice;

import pukhalskyi.entity.EventObject;
import pukhalskyi.cargoservice.kafka.consumer.OrderEventConsumer;
import pukhalskyi.cargoservice.kafka.producer.OrderEventProducer;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.producer.Producer;
```

```
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.KafkaException;
import org.apache.kafka.common.errors.ProducerFencedException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.time.Duration;
import java.util.concurrent.ExecutionException;


public class CargoService {
   private static final Logger LOGGER = LoggerFactory.getLogger(CargoService.class);
   private static final Consumer<Integer, EventObject> CONSUMER =
OrderEventConsumer.build();
   private static final Producer<Integer, EventObject> PRODUCER =
OrderEventProducer.build();

   public static void main(String[] args) {

      ConsumerRecords<Integer, EventObject> records;
      try {
         EventObject event;
         while (true) {
            records = CONSUMER.poll(Duration.ofMillis(100));
            for (ConsumerRecord<Integer, EventObject> record : records) {
               LOGGER.info("Consumed record in stock service method: Key {} Value {} "
+
                     "Partition {} Offset {}", record.key(), record.value(),
                  record.partition(), record.offset());
```

```
                event = record.value();
                event.setInStock(event.getProductId() > 0 && event.getAmount() > 0);
                event.setEvent("stock-check");
                publishStockCheckEvent(event);
            }
            CONSUMER.commitSync();
        }
    } catch (Exception ex) {
        LOGGER.error("An exception was thrown in stock service", ex);
    }
}

private static void publishStockCheckEvent(final EventObject event) {

    final ProducerRecord<Integer, EventObject> record =
            new ProducerRecord<>("STOCK_CHECK_EVENT_TOPIC",
event.getCustomerId(), event);

    try {
        PRODUCER.beginTransaction();
        final RecordMetadata metadata = PRODUCER.send(record).get();
        PRODUCER.commitTransaction();
        LOGGER.info("Stock check event published to Kafka: Topic {} Partition {} Offset
{}",
                metadata.topic(), metadata.partition(), metadata.offset());
    } catch (InterruptedException e) {
        LOGGER.error("An InterruptedException was thrown in publishStockCheckEvent
method", e.getMessage());
    } catch (ExecutionException e) {
```

```
            LOGGER.error("An InterruptedException was thrown in publishStockCheckEvent
method", e.getMessage());
        } catch (ProducerFencedException e) {
            LOGGER.error("An InterruptedException was thrown in publishStockCheckEvent
method", e.getMessage());
            PRODUCER.close();
        } catch (KafkaException e) {
            LOGGER.error("A KafkaException was thrown in publishStockCheckEvent
method", e.getMessage());
            PRODUCER.abortTransaction();
        }
    }
}
```

```
package pukhalskyi.cargoservice.kafka.producer;

import pukhalskyi.entity.EventObject;
import pukhalskyi.serializer.CustomSerializer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.IntegerSerializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Properties;
import java.util.UUID;
```

```
public class OrderEventProducer
{
package pukhalskyi.cargoservice;

import pukhalskyi.entity.EventObject;
import pukhalskyi.cargoservice.kafka.consumer.OrderEventConsumer;
import pukhalskyi.cargoservice.kafka.producer.OrderEventProducer;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.KafkaException;
import org.apache.kafka.common.errors.ProducerFencedException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.time.Duration;
import java.util.concurrent.ExecutionException;


public class CargoService {
    private static final Logger LOGGER = LoggerFactory.getLogger(CargoService.class);
    private static final Consumer<Integer, EventObject> CONSUMER =
OrderEventConsumer.build();
    private static final Producer<Integer, EventObject> PRODUCER =
OrderEventProducer.build();

    public static void main(String[] args) {
```

```
    ConsumerRecords<Integer, EventObject> records;
    try {
      EventObject event;
      while (true) {
        records = CONSUMER.poll(Duration.ofMillis(100));
        for (ConsumerRecord<Integer, EventObject> record : records) {
          LOGGER.info("Consumed record in stock service method: Key {} Value {} "
+
                  "Partition {} Offset {}", record.key(), record.value(),
              record.partition(), record.offset());

          event = record.value();
          event.setInStock(event.getProductId() > 0 && event.getAmount() > 0);
          event.setEvent("stock-check");
          publishStockCheckEvent(event);
        }
        CONSUMER.commitSync();
      }
    } catch (Exception ex) {
      LOGGER.error("An exception was thrown in stock service", ex);
    }
  }

  private static void publishStockCheckEvent(final EventObject event) {

    final ProducerRecord<Integer, EventObject> record =
        new ProducerRecord<>("STOCK_CHECK_EVENT_TOPIC",
event.getCustomerId(), event);
```

```
        try {
            PRODUCER.beginTransaction();
            final RecordMetadata metadata = PRODUCER.send(record).get();
            PRODUCER.commitTransaction();
            LOGGER.info("Stock check event published to Kafka: Topic {} Partition {} Offset
{}",
                    metadata.topic(), metadata.partition(), metadata.offset());
        } catch (InterruptedException e) {
            LOGGER.error("An InterruptedException was thrown in publishStockCheckEvent
method", e.getMessage());
        } catch (ExecutionException e) {
            LOGGER.error("An InterruptedException was thrown in publishStockCheckEvent
method", e.getMessage());
        } catch (ProducerFencedException e) {
            LOGGER.error("An InterruptedException was thrown in publishStockCheckEvent
method", e.getMessage());
            PRODUCER.close();
        } catch (KafkaException e) {
            LOGGER.error("A KafkaException was thrown in publishStockCheckEvent
method", e.getMessage());
            PRODUCER.abortTransaction();
        }
    }
}


package pukhalskyi.cargoservice.kafka.producer;

import pukhalskyi.entity.EventObject;
import pukhalskyi.serializer.CustomSerializer;
```

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.IntegerSerializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Properties;
import java.util.UUID;

public class OrderEventProducer
{
   private static final Logger LOGGER =
LoggerFactory.getLogger(OrderEventProducer.class);

   private OrderEventProducer() {}

   public static Producer<Integer, EventObject> build(){
      LOGGER.info("Initialize stock check event producer...");
      final Properties props = new Properties();
      props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
      props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
IntegerSerializer.class.getName());
      props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
CustomSerializer.class.getName());
      props.put(ProducerConfig.ACKS_CONFIG, "all");
      props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG,
UUID.randomUUID().toString());

      final Producer<Integer, EventObject> producer = new KafkaProducer<>(props);
```

```
        producer.initTransactions();
        return producer;
    }
}
```

```
package pukhalskyi.cargoservice.kafka.consumer;

import pukhalskyi.deserializer.CustomDeserializer;
import pukhalskyi.entity.EventObject;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.IntegerDeserializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Collections;
import java.util.Properties;

public class OrderEventConsumer
{
    private static final Logger LOGGER =
LoggerFactory.getLogger(OrderEventConsumer.class);

    private OrderEventConsumer() {}

    public static Consumer<Integer, EventObject> build() {
        LOGGER.info("Initialize order event consumer...");
        final Properties props = new Properties();
```

```
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
"orderEventConsumerGroup01");
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        final Consumer<Integer, EventObject> consumer = new KafkaConsumer<>(props,
                new IntegerDeserializer(),
                new CustomDeserializer<EventObject>(EventObject.class));

        consumer.subscribe(Collections.singletonList("ORDER_EVENT_TOPIC"));
        return consumer;
    }
}
```

Результат виконання програми (рис.3.6):

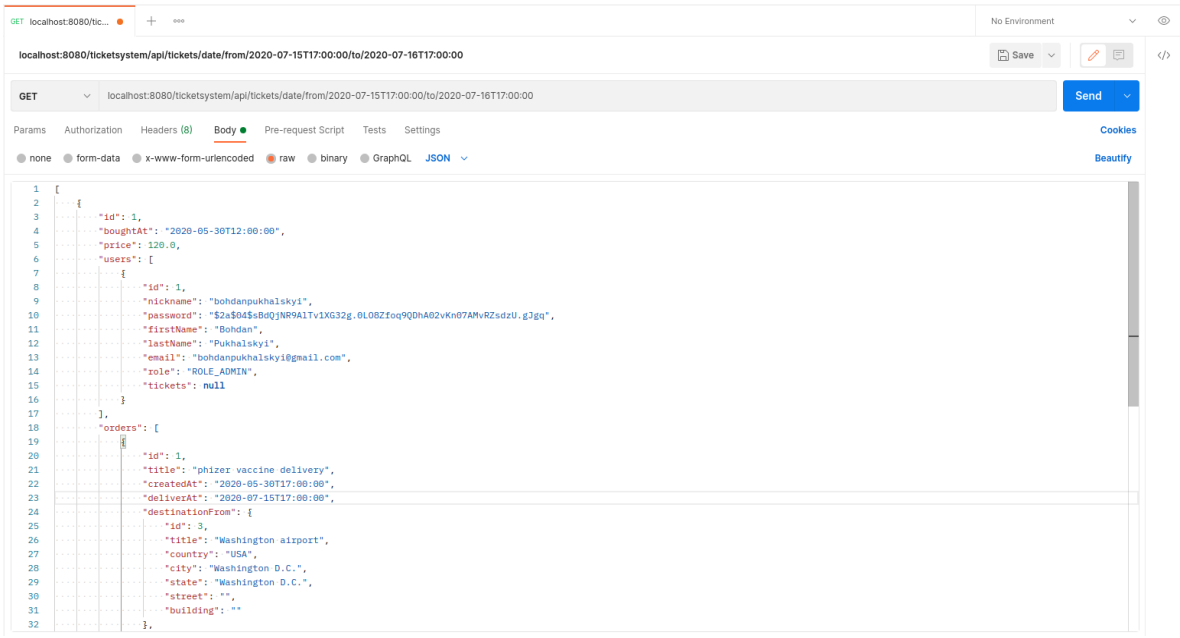

Рис. 3.6. Результат виконання платформи для замовлення вантажних авіаперевезень

3.2. Випробування на навантаження розробленої платформи

Випробування на навантаження – це тип тестування, при якому симулюється діяльність користувачів по заданому сценарію з метою визначення критерії продуктивності таких як час отримання відповіді, затримка, пропускна здатність та інші [29].

При даному дослідженні діяльність користувачів була симулювана інструментом випробування на навантаження Gatling. Gatling – це інструмент із відкритим програмним кодом, що дозволяє здійснити навантаження на систему[30]. Конфігурація даного інструменту є декларативною. Хоча сценарії реалізуються мовою програмування Scala, у більшості випадків достатньо лише модуля Gatling Recorder. Дана утиліта записує нашу активність як користувача, а згодом, генерує сам сценарій навантаження та записує його у вихідний файл із розширенням *.scala.

Наша конфігурація інструменту навантаження має наступний вигляд (рис. 3.7):

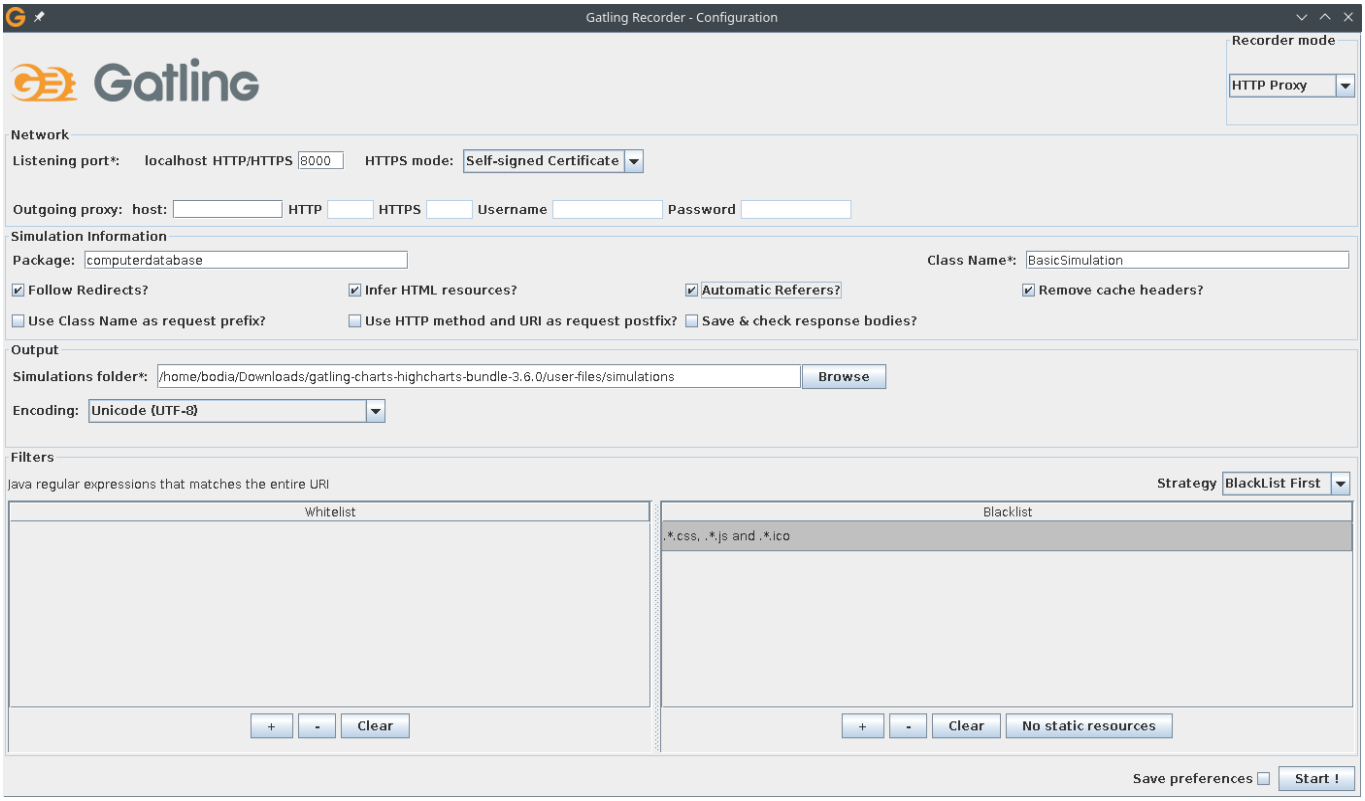

Рис. 3.7. Конфігурація Gatling

Все, що нам необхідно було зробити – вказати локальний хост та порт, а також відмітити опції “Follow redirects?”, “Infer HTML resources”, “Automatic Refers?” та вказати імена пакету та класу. Саме у цей клас і запишеться наш сценарій. Після натискання на кнопку “Start !”, утиліта починає записувати наші дії із застосунком. Також є і опція тегів, яка дозволяє нам, для зручності, помічати важливі операції.

Як було згадано раніше, розроблена програма містить дві версії. У першій запити здійснюються синхронно по протоколу HTTP. У другій – асинхронно, через Apache Kafka.

Розглянемо для початку першу версію. Існує багато критеріїв оцінки продуктивності системи. У нашому випадку, найважливішим є час отримання відповіді (англ. Response time). Саме цей критерій дає нам найзагальнішу оцінку швидкодії програмного продукту. На практиці даний етапи оцінки продуктивності є відправною точкою для подальшого аналізу. Графіки часу отримання відповіді мають наступний вигляд (рис. 3.8). Із наведених графіків видно, що швидкодія по часу отримання відповіді суттєво спадає від 143 мс. (32.35 % від загальної вибірки) до 559 мс (0,98 %). На 95 ому процентилі з часом, час отримання відповіді деградує від 557 мс. до 147 мс. Пропускна здатність варіюється від одного до двох запитів за секунду.

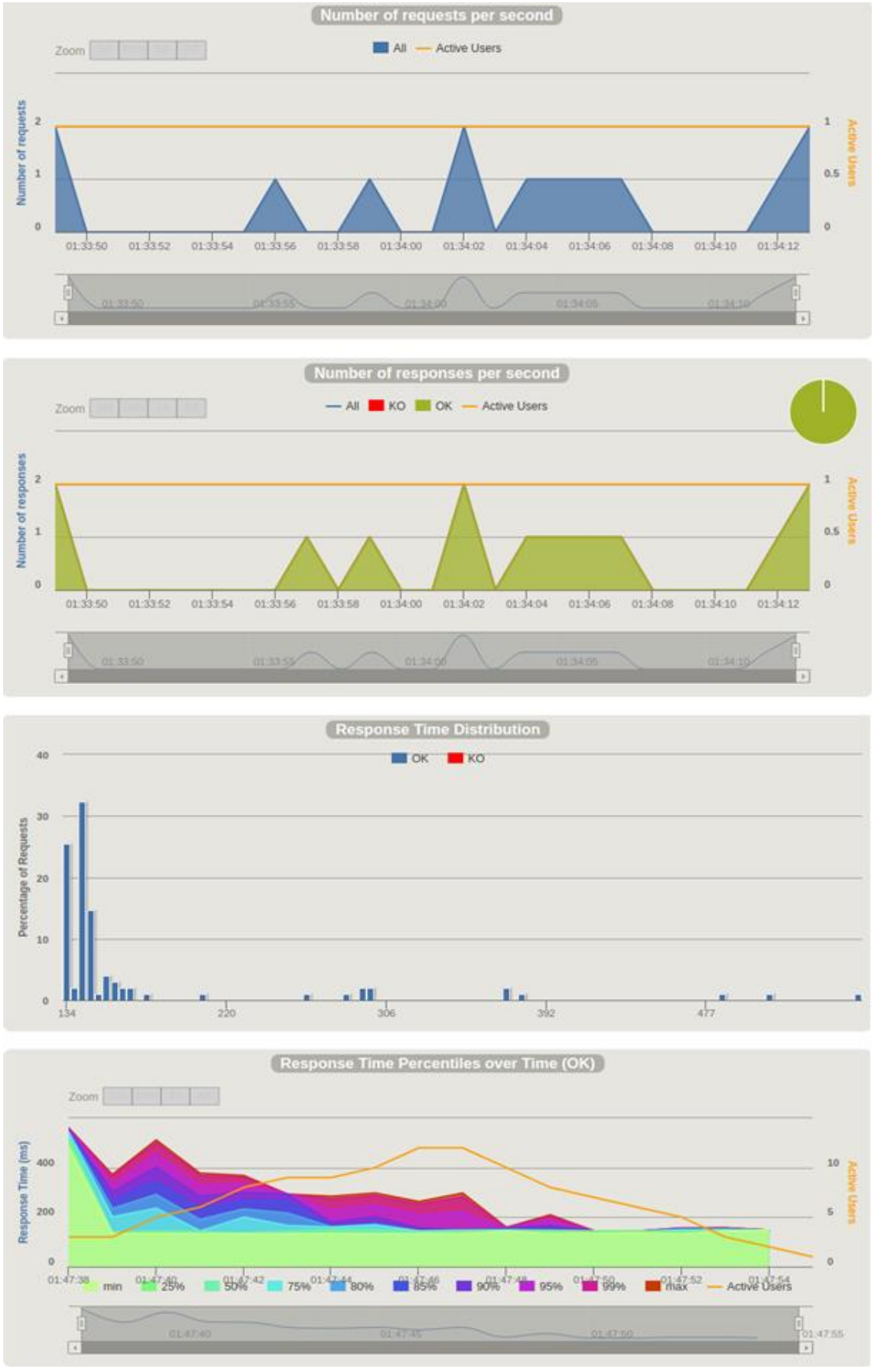

Рис. 3.8. Графіки динаміки часу отримання відповіді та його розподілення по процентилям для першої версії.

Результати тестування другої версії системи, що використовує брокер повідомлень наведено нижче (рис. 3.9).

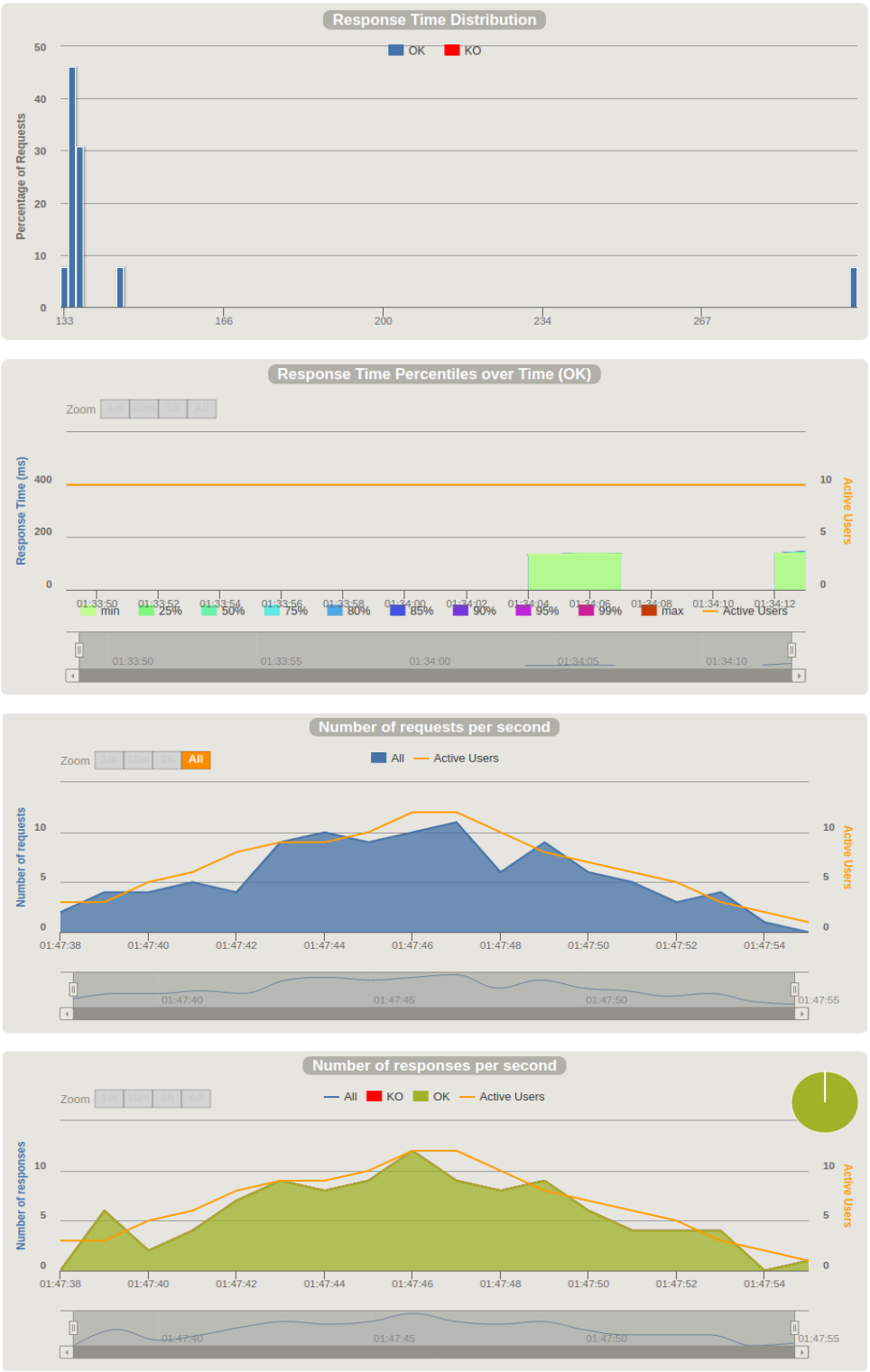

Рис. 3.9. Графіки динаміки часу отримання відповіді та його розподілення по процентилям для другої версії.

У системі із альтернативним способом передачі даних максимальний час отримання відповіді 135 мс. Якщо поглянути на час у 95 ому процентилі то він становить лише 137 мс., що є значно менше ніж у системі першої версії. Максимальна пропускна здатність становить одинадцять запитів за секунду.

3.3. Висновки до розділу

Після підключення системи обміну повідомленнями до розробленої платформи максимальний час отримання відповіді суттєво не скоротився, хоча якщо поглянути на вибірку даних по часу, а саме на 95 процентиль, що на практиці використовується найчастіше, час отримання відповіді зменшився у більш ніж 4 рази, а пропускна здатність зросла майже у шість раз. Це означає, що система працює продуктивніше протягом більш тривалого часу і як результат, користувач швидше отримує відповідь на свій запит.

Такий результат було досягнуто тим, що із підюченням брокеру повідомлень, при запиті користувача відбувається лише один віддалений виклик, при тому, що дані дістаються із трьох допоміжних сервісів. Досить часто для покращення часу отримання відповіді найоптимальнішим є зменшення віддалених викликів. Саме таке зменшення можна реалізувати за допомогою системи Apache Kafka, навіть у платформах зі складною бізнес логікою фільтрування та агрегації даних.

ВИСНОВКИ

Попит на інформаційні автоматизовані системи зростає з кожним днем. Все більше людей стають споживачами не лише “традиційної” продукції, а й віртуальної (потокове відео, музика і т.д.). Що вже і казати про запит замовників перевести бізнес в онлайн на тлі пандемії.

Зі зростанням кількості клієнтів та замовлень виникає потреба в структуруванні великих даних та їх оперативного опрацювання. Саме такі задачі виникають перед створенням систем реального часу, адже їх особливістю є доставка розрахунків до заданого терміну. Безліч методів було застосовано інженерами для забезпечення стабільної та оперативної роботи багатокористувацьких систем. Серед них: зменшення затримок, зменшення часу отримання відповіді користувачем, паралелізація відповідей, зменшення кількості перетворення даних та багато інших.

Існує багато засобів для реалізації перерахованих вище операцій. Деякі з них, дозволяють забезпечити роботу високонавантаженої системи та обробити дані дані у вказаний термін, хоча більшість, вже застаріла. З розвитком інформаційних технологій виникає проблема обробки нових моделей даних, які не є структурованими: починаючи із цифрових слідів користувачів для аналізу (пошукові запити) та закінчуючи мультимедіа (обробка облич). Системи, що застосовувались десятиліттями та непогано себе проявляли (реляційні бази даних, файлове сховище) вже не є придатними для зберігання та обробки даних, що досить часто зчитуються у наш час.

Дослідження даної проблематики починалося із аналізу “традиційних” способів передачі та зберігання даних, таких як: HTTP, різного роду бази даних, та сховища. Кожен із них володіє рядом своїх переваг та недоліків, але, навіть із тим, не відповідає вимогам систем реального часу. До особливостей таких систем належать: отримання опрацьованих даних до заданого терміну, передача інформації без втрат та відмовостійкість. Найкраще для вирішення таких питань підходить брокер повідомлень, який отримує дані із продюсерів та керує численними

споживачами, що зчитують із нього інформацію.

Зупинившись на брокері повідомлень, був розпочатий збір інформації про його реалізації та аналіз переваг і недоліків. Найбільш вживаними є три із них: Redis, RabbitMQ та Kafka. Перші два із них є досить простими у використанні, хоча не такими гнучкими у налаштуванні, а отже призначені для вузького спектру задач.

Наступним кроком була розробка платформи, що виконує аналіз потокових даних для опрацювання замовлень вантажних авіаперевезень та визначення термінів доставки. Архітектура системи була розроблена на базі чотирьох мікро сервісів, які забезпечують вищу доступність до кожного джерела даних, відмовостійкість та підтримку програмного забезпечення, а саме, версіонування.

Розроблена платформа була реалізована у двох версіях. У першій версії запити на той чи інший сервіс здійснювались синхронно через HTTP, а у другій - через Kafka. Згідно аналізу продуктивності двох варіантів, друга версія зарекомендувала себе краще.

Випробування на навантаження платформи двох версій відбувалось за допомогою інструменту навантаження систем Gatling. До уваги брались такі критерії продуктивності як час отримання відповіді та пропускна здатність. Час, за який відбувалась передача запиту від користувача та назад у другій версії виявився в чотири рази меншим ніж у першій, що дає нам зрозуміти про швидкодію вдосконаленої архітектури, а щодо пропускної здатності, то вона збільшилась майже в 6 раз.

Проведена робота була виконана відповідно до визначеної мети. Виконані всі завдання які поставлені під час розробки вимог до майбутньої системи, цілі були такі:

1. Дослідити предметну область.
2. Розглянути методи та засоби, що застосовуються при передачі повідомлень миттєво та без втрат між мікро сервісами.
3. Скласти порівняльну характеристику переваг та недоліків різноманітних брокерів повідомлень та альтернативних методів комунікації мікросервісів (протокол HTTP, спільна база даних, спільне сховище).

4. Впровадити обраний метод у інформаційну систему, яка аналізує та агрегує дані в реальному часі.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is the OSI Model? [Electronic resource] – Access mode: https://www.cloudflare.com/ (lastaccess: 19.05.21.). – Title from the screen.
2. HTTP request methods. [Electronic resource] – Access mode: https://developer.mozilla.org/(lastaccess: 20.05.21.). – Title from the screen.
3. Parameters and attributes in servlet. [Electronic resource] – Access mode: https://www.javajee.com/(lastaccess: 21.05.21.). – Title from the screen.
4. What Is Token-Based Authentication? [Electronic resource] – Access mode: https://www.okta.com/(lastaccess: 22.05.21.). – Title from the screen.
5. REST APIs. [Electronic resource] – Access mode: https://www.ibm.com/(lastaccess: 22.05.21.). – Title from the screen.
6. HTTP response status codes. [Electronic resource] – Access mode: https://developer.mozilla.org/(lastaccess: 22.05.21.). – Title from the screen.
7. Конноли Т. Базы данных. Проектирование, реализация и сопровождение. Теория и практика/ Т. Конноли – Москва : Вильямс, 2003. – 1440 с.
8. Select N+1 Problem. How to Decrease Your ORM Performance. [Electronic resource] – Access mode: https://dzone.com/(lastaccess: 23.05.21.). – Title from the screen.
9. Document Databases: How Do Document Stores Work? [Electronic resource] – Access mode: https://www.ionos.com/(lastaccess: 23.05.21.). – Title from the screen.
10. Object Storage. [Electronic resource] – Access mode: https://www.ibm.com/(lastaccess: 24.05.21.). – Title from the screen.
11. IBM Cloud Block Storage. [Electronic resource] – Access mode: https://www.ibm.com/(lastaccess: 24.05.21.). – Title from the screen.
12. Java Concurrency in Practice. / [Б. Ґоетс та ін.]. – 1st Publ. – Boston: Addison-Wesley Professional, 2006. – 432 p.
13. Real Time Systems. [Electronic resource] – Access mode: https://www.geeksforgeeks.org/(lastaccess: 25.05.21.). – Title from the screen.
14. Derui D. Performance Analysis and Synthesis for Discrete-Time Stochastic

Systems with Network-Enhanced Complexities / D. Derui, W. Zidong, W. Guoliang; Brunel University London. – Abingdon-on-Thames: Taylor & Francis Group, 2019. – 268 p.

15. Towards Turnkey Distributed Tracing. [Electronic resource] – Access mode: https://medium.com/(lastaccess: 25.05.21.). – Title from the screen.

16. Lou. W. A multipath routing approach for secure data delivery. Communications for Network-Centric Operations: Creating the Information Force./ W. Lou., Y. Fang. – Gainesville: Milcom, 2001. – 1467 p.

17. Amin S. Security of interdependent and identical networked control systems. Automatica./S. Amin, G. A. Schwartz, S. S. Sastry . – Santa Barbara: Elsevier, 2013. – 186 p.

18. What is DDoS mitigation? [Electronic resource] – Access mode: https://www.cloudflare.com/(lastaccess: 26.05.21.). – Title from the screen.

19. Sanjoy B. Multiprocessor scheduling for real-time systems./ Baruah S., Bertogna M., Buttazzo G.; University of California, Irvine Center for Embedded Computer Systems. – Irvine: Springer, 2015. – 234 p.

20. Deadline scheduling for real-time systems EDF and Related Algorithms / [J. A. Stankovic and oth.]. – New York: Springer Science and Business media, 2013. – 252 p.

21. Capacity augmentation bound of federated scheduling for parallel dag tasks. [J. Li and oth.]. – Madrid: Computer Sciences Commons, 2014. – 314 p.

22. Leung, J.Y.T. A new algorithm for scheduling periodic real-time tasks. / J.Y.T. Leung. – Irvine: Springer, 1989. – 377 p.

23. Narkhede N. Kafka: The Definitive Guide. / Narkhede N., Shapira G., Palino T. – Sebastopol: O'Reilly, 2019. – 197 p.

24. Bejeck W. P. Jr. Kafka Streams in Action Real-time apps and microservices with the Kafka Streams API. / W. P. Jr. Bejeck. – New York: Manning, 2018. – 280 p.

25. AMQP is the Internet Protocol for Business Messaging. [Electronic resource] – Access mode: https://www.amqp.org/(lastaccess: 27.05.21.). – Title from the screen.

26. Scott D. Kafka in Action. / Scott D., Gamov V., Klein D. – New York: Manning, 2017. – 375 p.

27. Mitra M. Mastering Gradle. /M. Mitra. – Birmingham: Packt Publishing, 2015. – 284 p.

28. Chacon S. Pro Git. /S. Chacon. – New York: Apress, 2014. – 440 p.

29. Fowler M. Patterns of Enterprise Application Architecture./ M.Fowler. – New York: Addison-Wesley Professional, 2002. – 560 p.

30. Load test as code [Electronic resource]. – Access mode: https://gatling.io/(lastaccess: 28.05.21.). – Title from the screen.